J-DSM : A Java Based Framework for Sharing Objects in a Distributed System

Yunus Emre Selcuk
Istanbul Technical University
Computer Engineering Department
Central Library, Istanbul, Turkey

Nadia Erdogan
Istanbul Technical University
Computer Engineering Department
Istanbul, Turkey

Abstract – This paper presents the design and implementation issues for the use of DSM techniques to allow shared access to objects in an object-oriented distributed environment. J-DSM supports sharing of Java objects, thus allowing a finer control over the granularity of sharing. J-DSM elaborates on two main classes of shared objects, mobile and stationary, which can be accessed through DSM and RMI mechanisms, respectively. Shared objects are uniquely named and manipulated through the J-DSM interface.

Keywords : distributed shared memory, dsm algorithms, consistency and coherence.

1. Introduction

Loosely-coupled distributed systems have evolved using message passing as the main paradigm for sharing information. Other paradigms used in loosely-coupled distributed systems, such as remote procedure call (RPC), are usually implemented on top of an underlying message-passing system. On the other hand, in tightly coupled architectures, the paradigm is usually based on shared memory for its simple programming model. The shared-memory paradigm has recently been extended for use in more loosely-coupled architectures and is known as *distributed* shared memory (DSM). DSM systems have many advantages over message-passing systems. Abstraction by DSM gives these systems the illusion of physically shared memory and allows usage of the shared-memory paradigm [1,2].

Most research done on DSM systems has concentrated on page-oriented shared memory. However, recent advances in DSM systems are providing increasing support for the sharing of objects rather than portions of memory [3]. This paper presents design and implementation issues for the use of DSM techniques to allow shared access to Java objects in an object oriented distributed environment. This allows a finer control over the granularity of sharing at the object level [4,5,6].

Taking a purely DSM-based approach can be limited. We have adopted a hybrid approach in J-DSM, where both DSM and RMI mechanisms are supported for object access, as function shipping and data shipping models of communication may be needed to develop efficient applications. J-DSM elaborates on two main classes of shared Java objects: *mobile shared-objects (mso)* use DSM mechanisms for. *Stationary shared-objects (sso)*, on the other hand, are remote objects and rely on RMI for being used. However, J-DSM hides all invocation details from user.

2. Design Outline

The current implementation consists of a set of abstract Java classes that manages both mobile and stationary shared-objects. A list of the important classes with their functionality is given in Figure 1.

DSMImpl	the server class that implements J-DSM on a node
DSMInfo	contains management info for each shared object
DSMExecutor	interprets request messages and takes approp.action
DSMPacketData	implements message format
DSMSpawnerInfo	keeps RMI management info for stationary objects
DSMSpawnerThread	responsible of stationary objects

Figure 1. Class architecture of J-DSM

Stationary shared-objects *do not need* replication. They are remote objects and are accessed through remote method invocations. Unlike mso's, sso's of the same class can be created with the same name on different nodes if they are parts of the solution of a distributed problem. A *Spawner Object* is responsible for the maintenance of all sso's of one class. The class that a sso belongs to is called a stationary DSM class and is derived from an ordinary user class. Calls to methods of this user class are invoked from the corresponding DSM class, considering current status of the sso. Certain remote methods explained in Section 5.2 are also added for management purposes. A preprocessor utility is also included which creates necessary management classes for sso's hiding remote invocation details from users.

2.1 Information Structure

We assume an implementation where a shared object directory is distributed among nodes that participate the J-DSM system. An entry of the directory contains management information for an object with following fields:

Name: Two processes in an application share an object if they call it by the same name.

Owner: The unique node which owns the only writable copy of the shared object.

Copyset: Set of nodes that have copies of a shared object.

Probable Owner: Each node keeps track of the probable owner of each shared object. This information provides a sequence of nodes through which the true owner is located.

Node List: Set of nodes currently participating in the DSM system.

Status: A shared object may be in one of the following states at any time:

readable : available and not locked
writable : available, replicas invalidated
available : present and contains valid data
locked : access denied except its owner

Spawner Table : A lookup table to find the spawner of a given stationary shared object.

3. The DSM Algorithm : Read-Replication

As we consider read sharing to be the characteristic of memory references in typical distributed applications, a *read-replication (multiple reader single writer)* [7] strategy is implemented to enable simultaneous accesses by different nodes to the same data and to minimize access latency. A write to a writable copy requires the use of other replicated copies be prevented. Therefore, we implement an *invalidation* based algorithm.

For providing *consistency and coherence*, a simple implementation of the *write-update* protocol is likely to be inefficient, because many replicas may be updated even if some of them are not going to be accessed in the near future. Therefore, we use the *write-invalidate* protocol for providing *release consistency.*

4. J-DSM Implementation

Management responsibility of shared objects is distributed to all nodes . The DSM algorithm is implemented by server processes present on each node. Figure 2 depicts the interactions between several components on a node. The functionality of each component is as the following.

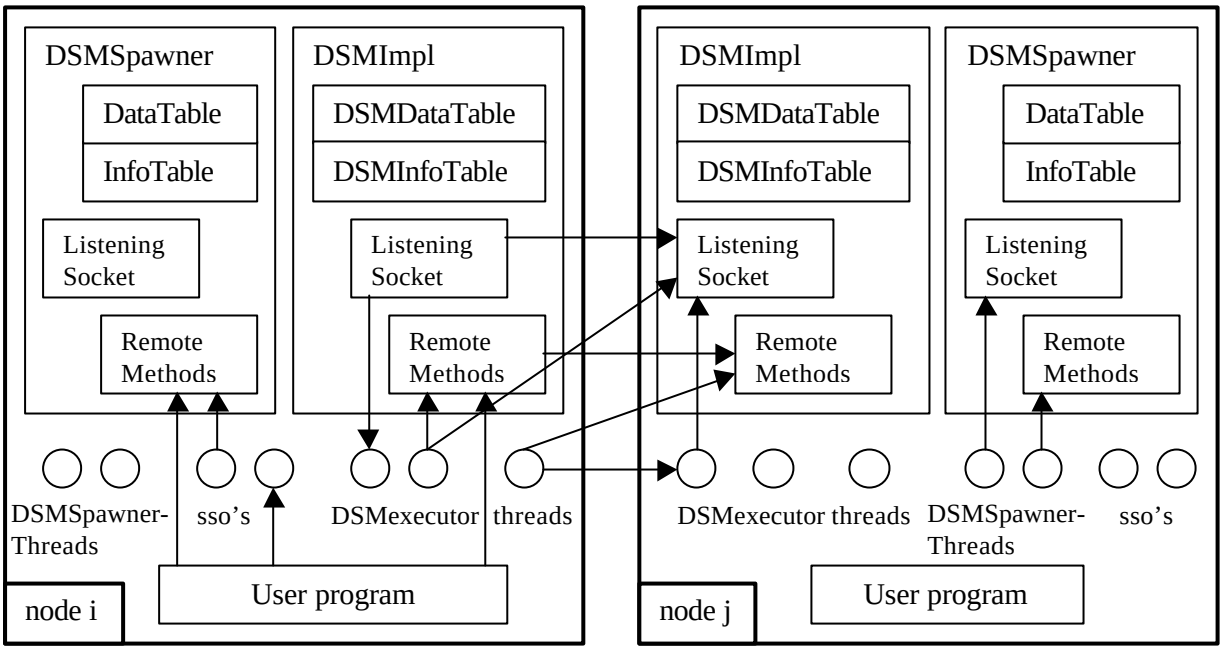

Figure 2 : Interactions between system components. A DSMSpawnerThread on node i can send messages to its counterpart on node j, as well as communicating with the DSMSpawner on node j.

DSM Server is responsible for maintenance actions of mso's. Current state of a mso and its management data are stored in two hashtables, DSMDataTable and DSMInfoTable. A DSM server process listens a UDP port for incoming messages. A new DSMexecutor thread is spawned to fulfill each request.

DSMexecutor threads resolve a received message and take necessary actions like replying to another DSM thread or calling a remote method of another DSM server.

Remote Methods that a DSM server supports are either called by other DSM servers on different nodes or construct the user interface, as explained in Section 5.

DSM Spawner is a simplified version of a DSM server, as an sso needs less management work than its mobile counterpart. Management data and current state of a sso is again stored in hashtables named DataTable and InfoTable. DataTable is only accessed before an object is being recreated or after it has been finalized.

Spawner Threads are similar to DSM threads, except executing fewer management commands.

6. J-DSM User Interface

Several remote methods constitute the user interface of J-DSM. Methods related to mso's are implemented by the DSMServer class, while those related to sso's are implemented by the DSMSpawner class and the sso itself. With the following intentions, user processes call the methods explained below.

6.1 Mobile shared-objects

Create : int CreateObject(...)

The class data and the unique name of the object to be created are supplied through the parameters. Local and remote directories are consulted to see if an object with the given name already exists. If not, an entry in the directory is allocated for the object and its ownership is assigned to the requesting process. A create error is returned, in case of any.

Remove : boolean RemoveObject(...)

A local mso is removed from the local node without looking at its status. No action is taken for its replicas on other nodes.

Read : Object ReadObject(...)

If a replica of the mso doesn't exist on the local node, the entire system is queried for the true owner of the object. This method returns a replica of the object for local use by the caller.

Read & Own : Object ReadLockObject (...)

This call returns a valid copy of the object together with its ownership. The message also contains an *invalidation request*.

Write : int ModifyObject(...)

Ownership of the object should be taken before its modification. If this is not the case, an access error is returned.

Lock/Unlock : Object LockObject (..)

This method includes two. The probable owner of the object is searched for within the DSM system with selective multicast messages sent to each node on the nodelist of the object, each by a new thread. The object is locked if the first flag is true, unlocked otherwise. If the object is to be locked and the second flag is true, the object is put in the unavailable state.

6.2 Stationary shared-objects

Create : DSMspawner.CreateOtonomus(...)

This method is similar to that of a mso, except for the addition of a boolean parameter. If a flag parameter is true, an object is created even if another object with the same name exists elsewhere. Replicas of a sso can only be created in this way.

Own/lock : DSMobject.DSMownOtonomus(...)

Invalidation request is handled if the block parameter is true. Otherwise, the method proceeds the way its counterpart does for a mso. Another method that should be implemented by a sso, *void transfer(...)*, is used within DSMownOtonomus method.

lock : DSMobject.DSMlockOtonomus(...)

This method is similar to that of a mso.

Read/write:

As a sso is not necessarily replicated, their methods are invoked remotely. Thus reading and updating members of a sso is possible only through remote calls. The user should add read/write methods to the sso class, if required.

7. RESULTS

This paper presents a flexible and portable Java framework, J-DSM, for distributed shared objects. The main feature of J-DSM is its support for both mobile and stationary shared objects., which improves the flexibility of the sytem. Users can choose between the two access protocols to shared objects to suit the application's semantics.

J-DSM is implemented fully in Java [7] and is portable to any platform that supports a JVM (Java Virtual Machine). The prototype implementation is tested on a network of NT 4.0 and Win9x platforms of Intel processors and initial results are encouraging. Future plans include work for improvements in performance and utilization of the framework for development of distributed applications.

References:

[1] B.Nitzberg and V.Lo, "Distributed Shared Memory:Asurvey of Issues and Algorithms", IEEE Computer, Vol.24, pp.52-60, Aug. 1991.

[2] M.Stumm and S.Zhou,"Algorithms Implementing Shared Memory", IEEE Computer, pp.54-64, May 1990.

[3]H.Bal, Programming Distributed Systems, Silicon Press, Prentice Hall, 1990.

[4] O. K.Sahingoz, "Implementation of a DSM System Based on Read Replication Algorithm", ITU, Ins. of Science and Tech., Ms.C. Thesis, 1998

[5] Y.E.Selcuk, "Implementatýon of a Distributed Shared Memory System", ITU,Ins. of Science and Tech., Ms.C. Thesis, 2000

[6] O.K.Sahingoz and N.Erdogan, "A Java Based DSM System for User Defined Shared Data Objects", Software and Hardware Engineering for the 21th Century, Editor N.E. Mastorakis, WSES Press, 1999

[7] G. Cornell, C.S.Horstmann, Core JAVA, Sun Microsystems Press, 1997.